Python Strings

Main Points

1. Create all kinds of strings; so many strings
2. Retrieve individual characters from strings
3. Slice (and dice?) your strings
4. Learn several, but not all, string methods

1 String Creation

Objects that contain alphabetic text characters are called *strings* and are used quite often in the programming world. Strings do not have to contain just letters. They can include nearly any character, such as letters, numbers, symbols, and even Unicode (like emoji or emoticons).

- Strings can be as short as single character
- Strings are defined by surrounding the text with a set of single or double quotes
  - Single quotes: `'parrot'`
  - Double quotes: `"Lumberjack"`
- How to include apostrophes and/or a set of quotes within strings...
  - Use an *escape sequence*
    - Include an apostrophe in a single quoted string with `\'`, i.e. `'isn\'t'`
    - Include double quotes in a double quoted string with `\"`, i.e. `"\"hello\""`
    - Include an actual back slash character with `\\` (if ever needed)
  - Use double quotes around strings with apostrophes in them, i.e. `"can't"`
  - Use single quotes around strings with double quotes in them, i.e. `'She said, "hello."'`
  - Enclose the string within three sets of single or double quotes, for example `""""It's only a flesh wound," he declared"""`
  - Three sets of single or double quotes can also be used to wrap multi-line strings

Double-quoted string

```
>>> name = "Brian Brady"
>>> name
'Brian Brady'

>>> print(name)
Brian Brady
```

Single-quoted string

```
>>> MET = 'Mechanical Engineering Technology'
>>> MET
'Mechanical Engineering Technology'
```

String with just numbers

```
>>> just_numbers = '8675309'
>>> just_numbers
'8675309'
```

Letters, numbers and special characters

```
>>> letters_nos_chars = "asdf1234!@#$"
>>> letters_nos_chars
'asdf1234!@#$'
```

String with an apostrophe in it

```
>>> apostrophe = "how's this"
>>> apostrophe
"how's this"
```

Phrase with double quotes

```
>>> short_phrase_double = '"Get to the chopper," he yelled.'
>>> short_phrase_double
'"Get to the chopper," he yelled.'
```

Sentence with an apostrophe and double quotes

```
>>> sentence = """HAL said "I'm sorry Dave, I can't do that." """
>>> sentence
'HAL said "I\'m sorry Dave, I can\'t do that." '

>>> print(sentence)
HAL said "I'm sorry Dave, I can't do that."
```

String spanning multiple lines

```
>>> three_liner = """Hello,
... is it me
... your looking for?"""

>>> three_liner
'Hello,\nis it me\nyour looking for?'

>>> print(three_liner)
Hello,
is it me
your looking for?
```

String with a line return between words

```
>>> two_words_newline = "Hello,\nWorld!"
>>> two_words_newline
'Hello,\nWorld!'

>>> print(two_words_newline)
Hello,
World!
```

Three words with two tabs between each word

```
>>> three_with_tabs = "One\t\tTwo\t\tThree"
>>> print(three_with_tabs)
One         Two         Three
```

Unicode characters in a string

```
>>> # use \u for 8 bit like \u00b2 for squared
>>> # use \U for 16 bit like \U0001F600 for a smiley face
>>> # go to https://home.unicode.org to see them all

>>> squared_cubed = 'inch\u00b2 and inch\u00b3'
>>> print(squared_cubed)
inch² and inch³
```

The `len()` function returns the number of characters in a string; the string's length. For example, `len("Python")` will tell you that the string is 6 characters long. The function can also return the length (number of items) in lists and tuples (as will be seen in another document).

Finding a string's length

```
>>> funny_quote = "Nobody expects the Spanish Inquisition!"
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!
>>> len(funny_quote)
39
```

2 Accessing Individual String Characters

Strings are lists of characters grouped together but treated as one object. Specific characters in strings can be accessed based on their position in the string. The process of accessing single characters is referred to as "*string indexing*". It is very important to note that all counting in *Python* starts with zero, not one.

- Place the desired index number enclosed in square brackets after a string or string name
- `"H"` in the string `"Hello"` is in the zeroth position (the zeroth index)
- Access the character at the zeroth index with `"Hello"[0]`
- Access the first `l` using `"Hello"[2]`
- Using `my_string[3]` will access the character located at index position 3 (the fourth character)
- The last item in `my_string` can be accessed using `my_string[len(my_string) - 1]`

The following examples use string indexing to access specific characters from `funny_quote`.

Uppercase "S" in "Spanish"

```
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> funny_quote[19]
'S'
```

The "!" at the end using the `len()` function

```
>>> funny_quote[len(funny_quote) - 1]
'!'
```

The letter "x"

```
>>> funny_quote[8]
'x'
```

The letter “q”

```
>>> funny_quote[29]
'q'
```

The following function, `string_ruler(message)`, was created to check the index positions of any character in a string 100 characters long or less. Don't worry how it works at this point in time; just use it if you need it.

A string ruler function

```
def string_ruler(message):
    new_message = spacer = '|'
    left_index_top = '|'
    left_index_bottom = '|'
    for i, letter in enumerate(message):
        new_message += letter + '|'
        spacer += ' |'
        if i > 9:
            left_index_bottom += str(i%10) + '|'
            left_index_top += str(i//10) + '|'
        else:
            left_index_bottom += str(i) + '|'
            left_index_top += ' |'
    print(f'string: {new_message}')
    print(f'        {spacer}')
    print(f'  left  {left_index_top}')
    print(f' index: {left_index_bottom}')

string_ruler("Nobody expects the...")
```

```
string: |N|o|b|o|d|y| |e|x|p|e|c|t|s| |t|h|e|.|.|.|
        | | | | | | | | | | | | | | | | | | | | |
  left  | | | | | | | | | | |1|1|1|1|1|1|1|1|1|1|2|
 index: |0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
```

- Strings can also be indexed from the right instead of the left
- Indexing from the right uses negative index numbers
- Index of the first character from the left is `[0]`
- Last character in a string can be indexed directly using `[-1]`
- Negative sign indicates that indexing is from the right end of the string
- Indexing from the right starts with `-1` not `-0`, because using -0 would likely cause confusion since `0` and `-0` are equal

String indexing diagram

```
| 0 | 1 | 2 | 3 | 4 | <-- Indexing from the left
|   |   |   |   |   |
| H | e | l | l | o | <-- String characters "Hello"
|   |   |   |   |   |
|-5 |-4 |-3 |-2 |-1 | <-- Indexing from the right
```

The following examples use indexing from the *right* to access each of the requested characters from `funny_quote`.

The letter “S” using negative indexing (i.e. from the right)

```
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> funny_quote[-20]
'S'
```

The character “!” using negative indexing

```
>>> funny_quote[-1]
'!'
```

The letter “x” using negative indexing

```
>>> funny_quote[-31]
'x'
```

The letter “q” using negative indexing

```
>>> funny_quote[-10]
'q'
```

The first letter using negative indexing and `len()`

```
>>> funny_quote[-len(funny_quote)]
'N'
```

3 Accessing Portions of Strings, aka Slicing

Multiple characters from a string, a sub-string, can be accessed via *string slicing*. To slice a string, square brackets are placed after the string or string name with the slicing information, similar to how indexing works.

- Square brackets include the start, stop, and step values separated by colons
  - The slice includes the character at the start index
  - The slice ends one character before the end index
- All three of the slice arguments are optional
  - If no step, it is assumed to be `1`
  - If no start, it is assume to be...
    - `0` for positive steps
    - The last character for negative steps
  - If no end, the last character included is assumed to be...
    - The last character of the string for positive steps
    - The first character for negative steps
- The slice `[:]` will return a copy of the entire string
- The diagram below illustrates how slice numbering works; slices actually occur between characters
- Some examples...
  - `my_string[0:3]` and `my_string[:3]` and `my_string[0:3:1]` are the same
  - `my_string[0:]` and `my_string[0::1]` and `my_string[:]` are the same
  - `"Hello"[0:3]` would return `"Hel"`
  - `"Hello"[2:]` would return `"llo"`

String slicing diagram

```
 0   1   2   3   4   5 <-- Slice from left; my_string[1:4] => 'ell'
 |   |   |   |   |   |
 | H | e | l | l | o | <-- String characters "Hello"
 |   |   |   |   |   |
-5  -4  -3  -2  -1   | <-- Slice from right with a positive step;
 |   |   |   |   |   |       my_string[-4:-1] => 'ell'
 |   |   |   |   |   |
-6  -5  -4  -3  -2  -1 <-- Slice from right with a negative step;
                             my_string[-1:-5:-1] => 'olle'
```

The following examples return sub-strings from the `funny_quote` string.

1. The last 12 characters
2. Every other character from the left to the right
3. The entire string written backwards (think for a second)
4. From 19 to 26

Slicing the first 6 characters

```
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> funny_quote[0:6]
'Nobody'

>>> funny_quote[:6]
'Nobody'
```

Slicing the last 6 characters

```
>>> funny_quote[-6:]
'ition!'
```

Slicing every other character from left to right

```
>>> funny_quote[::2]
'Nbd xet h pns nusto!'
```

Slicing the entire quote backwards

```
>>> funny_quote[::-1]
'!noitisiuqnI hsinapS eht stcepxe ydoboN'
```

Slicing from index 19 to 26

```
>>> funny_quote[19:26]
'Spanish'
```

4 String Methods

There are quite a few methods that can act on or with strings. Use `x.my_method()` to work with or on an object `x` using a method called `my_method()`. The following code cell prints a list of string methods, functions, and operations. The methods are the items without leading and trailing underscores Using `print(dir(str))` will also print all of the string methods.

String methods list

```
>>> my_string = "Hello, World!"
>>> print(dir(my_string))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 ↪  '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 ↪  '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 ↪  '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
 ↪  '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 ↪  '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 ↪  '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
 ↪  'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
 ↪  'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
 ↪  'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
 ↪  'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
 ↪  'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
 ↪  'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
 ↪  'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
 ↪  'translate', 'upper', 'zfill']
```

The following code will create a cleaner looking list of methods. This technique uses a list comprehension that will be covered at another time.

Concise list of string methods

```
>>> print([item for item in dir(str) if "__" not in item])
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
 ↪  'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
 ↪  'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
 ↪  'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
 ↪  'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 ↪  'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
 ↪  'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 ↪  'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Not all of these methods will be investigated in this document, only some of them. Specifically, methods related to modifying strings and methods used for counting, searching, and replacing parts of strings will be reviewed.

4.1 Methods for Modifying Strings

4.1.1 Immutability of Strings

Python strings are *immutable*. This means that strings cannot be changed (mutated or modified) after they have been created. To “modify” a string, a copy must be created that has the desired changes. Methods that “modify” strings create copies and the original strings remain unchanged. The copies that are created contain the modifications. The original string can be replaced by assigning the newly created string to the original variable name.

4.1.2 Methods for Changing Case

The methods included here are used for changing the case of strings. The names of these methods essentially describe what they do.

- `str.lower()`
- `str.upper()`
- `str.title()`
- `str.swapcase()`
- `str.capitalize()`

All lowercase

```
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> print(funny_quote.lower())
nobody expects the spanish inquisition!

>>> print(funny_quote)   # Notice that the original does not change
Nobody expects the Spanish Inquisition!
```

All uppercase

```
>>> print(funny_quote.upper())
NOBODY EXPECTS THE SPANISH INQUISITION!
```

First letter of only the first word only is capitalized

```
>>> print(funny_quote.capitalize())
Nobody expects the spanish inquisition!
```

Swap upper and lowercase in the string

```
>>> print(funny_quote.swapcase())
nOBODY EXPECTS THE sPANISH iNQUISITION!
```

First letter in each word is uppercase

```
>>> print(funny_quote.title())
Nobody Expects The Spanish Inquisition!
```

4.2 Methods for Adding/Removing Spaces

Several string methods essentially deal with adding or removing spaces (and sometimes other characters) to/from the left and right ends of strings. These can be used for cleaning up text, especially user input.

- `str.center(width)`, `str.ljust(width)`, and `str.rjust(width)`
  - Returns the original string within a new string of length `width`
  - Original string will be centered, left justified, or right justified
  - Spaces are used as filling (padding) by default
  - An optional single-character string argument can be included for padding instead of spaces
- `str.strip()`, `str.lstrip()`, and `str.rstrip()`
  - Strips whitespace from both ends, the left end, or the right end of the string
  - Includes `\n` and `\t` characters
  - Accepts a string of any/all characters to strip as an optional argument
  - `my_string.strip(' !?')` will remove spaces, !, and ? from both ends of `my_string`

Center a string within the original string length plus 20 characters

```
>>> spam = "Spam, spam, eggs, and spam"
>>> spam.center(len(spam) + 20)
'          Spam, spam, eggs, and spam          '

>>> # now with periods as padding
>>> spam.center(len(spam) + 20, ".")
'..........Spam, spam, eggs, and spam..........'
```

Left and right justify a string with character padding

```
>>> "Lumberjacks".ljust(20, "?")
'Lumberjacks?????????'

>>> "Parrots".rjust(20, "_")
'_____________Parrots'
```

Stripping whitespace from both ends of a string

```
>>> "\t\t  Hello  \n".strip()
'Hello'
```

Strip spaces, periods, question marks, and exclamation marks from the right end of a string

```
>>> "What????....!!!!!    ".rstrip(" .?!")
'What'
```

4.3 Methods for Counting, Searching, and Replacing

- `str.count(sub, start, end)`
  - Count the number of times that a sub-string appears in a larger string
  - Requires at least one argument; the sub-string to be counted (a string or the name of a string)
  - Can include two optional arguments...
    - Index position at which to start counting
    - Index immediately after the index position at which to end counting
    - It is not possible to only specify an ending index
  - Examples
    - `my_string.count("a")` returns the number of times `"a"` occurs in `my_string`
    - `my_string.count("a", 2, 7)` counts starting at index 2 and ends just before index 7
    - `my_string.count("a", 4)` counts from the 4th index to the end of the string
- `str.find(sub, start, end)`
  - Use `string_a.find(string_b)` to look for `string_b` within `string_a`
  - If found, returns the index from `string_a` matching the `string_b` starting position
  - If not found, returns `-1`
  - If `string_b` occurs more than once, only the position of the first occurrence will be returned
  - Accepts optional starting and ending arguments like `str.count()`
  - Good idea to check if the sub-string is in the string first; i.e. `"i" in "team"`
  - `my_string.find("the")` returns the index of the first occurrence of `"the"`
- `str.replace(old, new, count)`
  - `str_a.replace(str_b, str_c)` replaces all occurrences of `str_b` with `str_c` in the string `str_a`
  - Two required and one optional argument
    1. Search string (required)
    2. New string (required)
    3. Number of occurrences to replace (optional)
  - `my_string.replace('I', 'you', 2)` replaces the first 2 occurrences of `'I'` with `'you'`
  - The original string is not changed; use `my_string = my_string.replace(old, new)` to "modify" the original string

Most of the following examples use the string `funny_quote`. The original string is never changed since the "modified" strings are never reassigned to the string name.

How many times does "e" appear in `funny_quote`?

```
>>> funny_quote.count("e")
3
```

How many times does “is” appear?

```
>>> funny_quote.count("is")
2
```

How many times does “o” appear starting at index position 15?

```
>>> funny_quote.count("o", 15)
1
```

What is the location of the first “the” in the string?

```
>>> funny_quote.find("the")
15
```

Find the position of “x” starting at index 10

```
>>> funny_quote.find("x", 10)
-1
```

Replace “Spanish” with “Ferris”

```
>>> funny_quote.replace('Spanish', 'Ferris')
'Nobody expects the Ferris Inquisition!'
```

Replace every “i” with “I”

```
>>> funny_quote.replace('i', 'I')
'Nobody expects the SpanIsh InquIsItIon!'

>>> print(funny_quote)
Nobody expects the Spanish Inquisition!
```

Reassigning a modified string back to the original variable

```
>>> greetings = "My name is Brian"
>>> print(greetings)
My name is Brian
>>> greetings = greetings.replace("Brian", "Mr. Brady")
>>> print(greetings)
My name is Mr. Brady
```

4.4 Methods for Checking the Starting/Ending Characters

The following methods return `True` or `False` instead of a modified string. Same as elsewhere in *Python*, the case of string characters matter.

- `str.startswith(prefix)` – returns `True` if the original string starts with the string `prefix`
- `str.endswith(suffix)` – returns `True` if the original string ends with the string `suffix`

Check the starting characters of the string

```
>>> spam = "Spam, spam, spam, and eggs"
>>> spam.startswith("spam")
False

>>> # change the original string to lowercase first...
>>> spam.lower().startswith("spam")
True
```

Check the ending characters of the string

```
>>> spam.endswith("eggs")
True
```

4.5 The "is" String Methods

These methods ask a question of the string or a portion of a string on which they act.

- Partial list of the `"is"` string methods
  - `str.islower()` – Are all characters lowercase?
  - `str.isupper()` – Are all characters uppercase?
  - `str.isspace()` – Are all characters spaces?
  - `str.istitle()` – Does each word start with an uppercase letter with the rest lowercase?
  - `str.isalpha()` – Are all characters alphabetic?
  - `str.isalnum()` – Are all characters either alphabetic or numeric?
  - `str.isnumeric()` – Are all characters numeric (0-9)?
- If the answer to the question is true, then the method returns a Boolean `True` value
- If the answer to the question is false, then the method returns a Boolean `False` value
- `my_string.isalpha()` asks if the entire string is composed soley of alphabetic characters
  - If it does, then the method returns a Boolean `True` value
  - If not, the method returns a Boolean `False` value
- Portions of strings can be used via slicing
  - `my_string[:3].islower()` only checks the first 3 characters
  - `my_string[-4:].isnumeric()` only checks the last 4 characters

Several of the following code blocks demonstrates using the `"is"` methods on portions of `funny_quote`. The remaining blocks use strings containing numeric values.

Is the slice [0:6] upper case?

```
>>> print(funny_quote[:6])
Nobody
>>> funny_quote[:6].isupper()
False
```

Is the slice [0:6] title case?

```
>>> funny_quote[:6].istitle()
True
```

Is the slice [7:14] lower case?

```
>>> print(funny_quote[7:14])
expects
>>> funny_quote[7:14].islower()
True
```

Are index positions 6 and 10 spaces?

```
>>> funny_quote[6].isspace()
True

>>> funny_quote[10].isspace()
False
```

Is the slice [0:6] alphabetic?

```
>>> # Is 0 to 6 alphabetic?
>>> funny_quote[:6].isalpha()
True
```

Is the slice [7:14] alpha-numeric?

```
>>> funny_quote[7:14].isalnum()
True
```

Are the strings ‘42’ and ‘42.0’ numeric?

```
>>> '42'.isnumeric()
True
```

```
>>> '42.0'.isnumeric()
False
>>> # the decimal point is not numeric, therefore False
```

5 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapter 8
- *The Coder's Apprentice*, Pieter Spronck, chapter 10
- *A Practical Introduction to Python Programming*, Brian Heinold, chapter 6
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapter 9